How to write an ATC Check

A brief introduction.

EXTERNAL

The ATC is an important framework for carrying out static checks of ABAP source code and ABAP dictionary objects. The remote scenario allows a check system running the newest release to run its checks against a system running an older release.

In the following, you will learn how to write your own check to analyze source code and how to ensure it is able to be carried out remotely.

TABLE OF CONTENTS

PRELIMINARIES

Please note that this guide assumes you are working with a system with SAP release 7.54 or higher. Some of the features and specifics described work differently or are not present at all in lower releases.

The first chapters of this guide document various aspects of creating an ATC check. If you prefer learning by example, skip to the chapter An example: Detecting obsolete KEYWORDS.

CREATING A CHECK CLASS

The first step is to create a class that the framework will recognize as a new check. You need to create a class that inherits from any class that is a subclass of `CL_CI_TEST_ROOT`. By convention, check classes start with `CL_CI_TEST` (or `ZCL_CI_TEST` in customer namespace). By convention, check classes start with `CL_CI_TEST` (or `ZCL_CI_TEST` in customer namespace). Note that result classes conventionally start with `CL_CI_RESULT`, so it is a good idea to use a suffix for your check class that does not exceed the maximum of 30 characters when appended to `CL_CI_RESULT`.

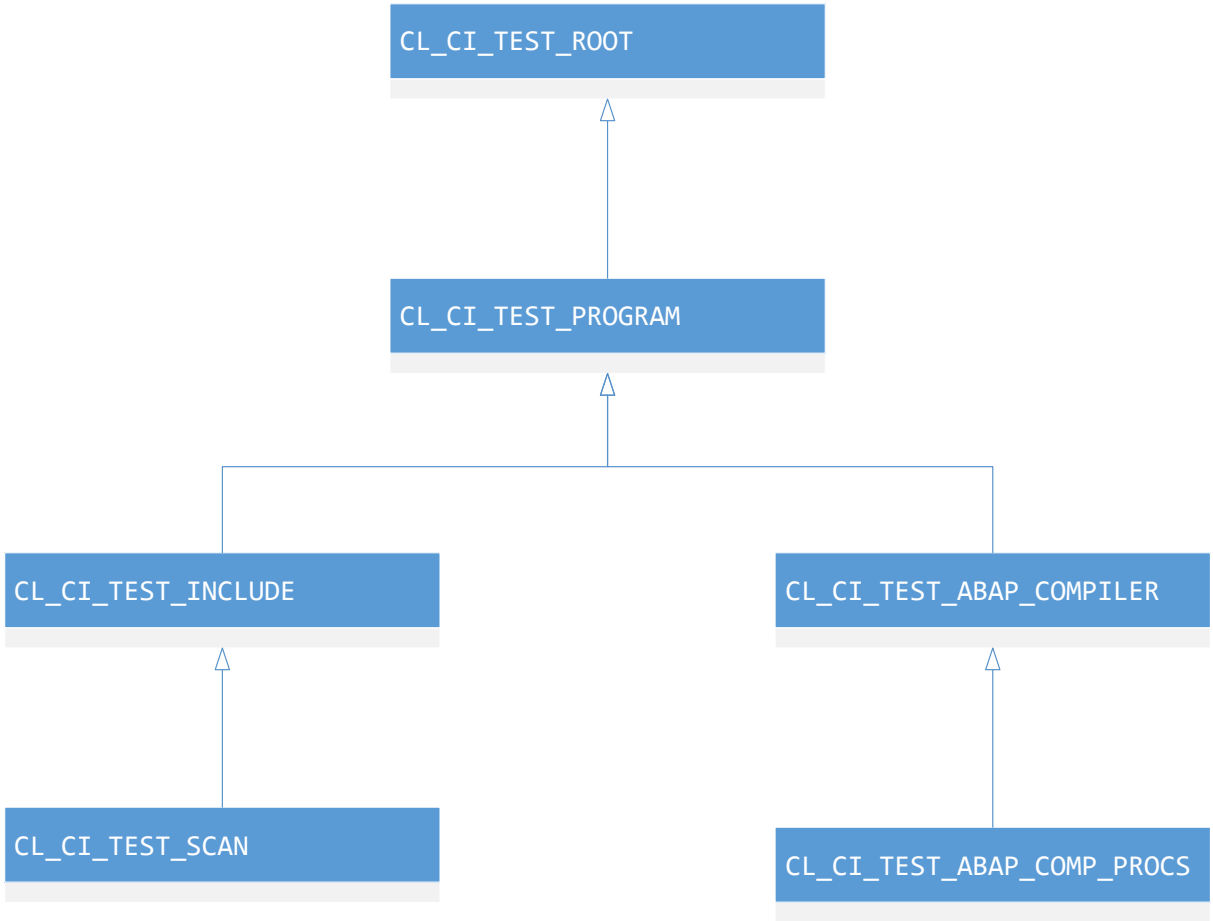

Figure 1: UML diagram of check classes. All check classes must inherit from CL_CI_TEST_ROOT. To reuse existing data collection routines, tests may inherit from CL_CI_TEST_SCAN or CL_CI_TEST_ABAP_COMP_PROCS to analyze ABAP source code.

When asked to carry out your check, the framework will instantiate an object of your check class and then call its methods `RUN_BEGIN`, `RUN` and `RUN_END`. Therefore, the main logic of your check needs to be implemented in these methods. The way this implementation proceeds depends on what kind of check exactly you have. Your three main options are:

1. A subclass of `CL_CI_TEST_ABAP_COMP_PROCS`
2. A subclass of `CL_CI_TEST_ROOT`

3. A subclass of CL_CI_TEST_SCAN

While a lot of implementation details depend on which of these options you have chosen, there are a few things that always work in the same manner.

1. Checks based on CL_CI_TEST_ABAP_COMP_PROCS

Checks based on CL_CI_TEST_ABAP_COMP_PROCS are meant to analyze tokenized ABAP source code. The paradigm in this case is to partition the code into programming blocks – simply called “procs”, which is short for “procedure” – and analyze the code proc by proc. If the main goal of your check is to analyze ABAP statements, then this is the recommended choice.

A programming block is a logically coherent unit of source code. In the context of ABAP Objects, this mainly refers to the declaration and implementation blocks of classes, with each implementation block itself split up into one block for each method implementation. For reports, a programming block is basically equivalent to an event block, i.e. the set of statements following a START-OF-SELECTION, AT-SELECTION-SCREEN or any other event reporting keyword. If a report contains multiple START-OF-SELECTION words, this will lead to multiple programming blocks in the CL_ABAP_COMP_PROCS output, even though logically these form a single set of statements all executed when the event occurs.

2. Checks based on CL_CI_TEST_ROOT

Any ATC check must inherit from CL_CI_TEST_ROOT to be properly detected by the framework. For checks that examine ABAP source code, enhanced source information is provided when inheriting from CL_CI_TEST_ABAP_COMP_PROCS.

We generally do not recommend directly inheriting from CL_CI_TEST_ROOT. However, if your test does not check ABAP source code and there is no more suitable superclass for the objects you want to check, this is a valid option. If you are writing several checks that are based on the same information, consider writing a common superclass that streamlines the collection of this information. (Of course, it is not forbidden to use composition instead of inheritance to re-use data collection routines.)

3. Checks based on CL_CI_TEST_SCAN

With the advent of CL_CI_TEST_ABAP_COMP_PROCS, there is generally no reason to rely on the raw output of SCAN ABAP-SOURCE that CL_CI_TEST_SCAN provides. However, there are some edge cases, such as checks that are intended to process arbitrary comments in the source code, where the SCAN architecture is unavoidable. There already exists a brief description on how to write checks in the CL_CI_TEST_SCAN framework on the SCN wiki (link to PDF) and the SAP blog community.

Moreover, the book "SAP Code Inspector" published by Galileo Press contains relevant information for checks based on CL_CI_TEST_SCAN (ISBN: 978-3-8362-1706-4).

IMPLEMENTING YOUR CHECK

To obtain a check that can be executed, there are a few minimal implementation steps required so that the framework knows how to execute your check and display its results:

Setting up attributes of your check

The initialization of the important attributes of your check is done in the check class’s instance constructor. You should call the constructor of its superclass, if applicable, and then adjust the attributes as appropriate for your check. A typical constructor might look like this:

```
METHOD constructor .

  super->constructor( ).
  description = 'My class description'(001).
  category    = 'CL_CI_CATEGORY_MY_CATEGORY'.
  position    = '001'.
  version     = '000'.
  has_attributes      = abap_true.
  attributes_ok       = abap_true.
  has_documentation   = abap_true.
  remote_enabled      = abap_true.
  remote_rfc_enabled  = abap_true.
  uses_checksum       = abap_true.
  check_scope_enabled = abap_true.
```

- `DESCRIPTION` contains a short text description of the purpose of your check, which will be displayed in the SCI transaction when creating a check variant.
- `CATEGORY` is the technical name of the category (e.g. performance checks, S/4HANA readiness checks, etc.) your check falls into. A category is technically a class that inherits from `CL_CI_CATEGORY_ROOT`. To get the technical names of an already existing category, go to the dynpro for creating and displaying check variants in the SCI transaction (reached via the bottom-most of the three entry fields on the main screen of the transaction) and choose the menu item *Checkvariant->Display->Technical Names* there.
- `POSITION` is a numeric string that controls at which position your check is displayed within its category. The topmost check will be the one with the lowest (positive) value for position, the bottommost the one with the largest. If two checks have the same value for position, it is undefined which of them will be displayed first.
- `VERSION` is a numeric string that allows both you and the framework to identify if a check has changed substantially. Whenever you make an edit to your check that changes it in an incompatible way, you should increase this number. Check variants that include a test whose version has changed since it was created cannot be executed anymore. Consequently, the version should not be changed too often since check variants need to be redefined after each version change. But it is a advisable to increase the test version if the results of the check are significantly different. In that case, increasing the version forces all users to re-evaluate whether they want to run the check.
- `HAS_ATTRIBUTES` informs the framework if your check has attributes that a user can or must set before running it.
- `ATTRIBUTES_OK` tells the framework whether or not the default values for the settings of your check are suitable for its execution. If false, the framework will reject attempts to execute the check in a variant without its settings being adjusted manually.
- `HAS_DOCUMENTATION` informs the framework if your check has attached documentation that can be displayed. You should always document your check! This documentation should be created via the SE61 transaction, see the section “Creating documentation” below.
- `REMOTE_ENABLED` allows to check with an outdated scenario (which uses a push approach rather than a pull approach by manually uploading the extracted source code to be checked). If your check is based `CL_CI_TEST_ABAP_COMP_PROCS` and the flag `REMOTE_RFC_ENABLED` is set to `ABAP_TRUE`, you can also set `REMOTE_ENABLED` to `ABAP_TRUE`.
- `REMOTE_RFC_ENABLED` indicates that your check can be carried out remotely. Note that it is your responsibility to ensure that your check can run remotely if you set these attributes as true – the attributes themselves are merely of an organizational nature and do not enable your test to run remotely! Additionally, beware that these attributes might be inherited from the superclass if you do not set them yourself, even if the classification does not apply to your check.
- `USES_CHECKSUM` means that your check will generate a checksum for each finding to identify its location. This is relevant, for instance, when you want to create ATC exemptions for a finding: The checksum is used to ascertain that the environment of the finding has not changed compared to the state when the exemption was issued. When setting this attribute to `ABAP_TRUE`, you are entering a binding contract: You promise that your check will not emit any findings that do not have a checksum

attached (the values 0 and -1 for the checksum correspond to “no checksum”). The framework will generally react to a violation of this contract by aborting the check run (directly, via an exception or via an assertion).

- `CHECK_SCOPE_ENABLED` The default setting should be `CHECK_SCOPE_ENABLED = ABAP_FALSE`. In this case, findings in SAP code will not be reported in customer systems (but findings in modifications of SAP code will, if the corresponding ATC settings are activated). If you set this parameter to true, you claim that your check already evaluates whether the findings are “in scope”, i.e. caused by customer code and not SAP code or generated code.

Attributes you do not set in the constructor are inherited from its superclasses, if any of them sets the attribute in question in its own constructor. If no superclass sets an attribute, it defaults to its initial value.

Registering checked object types

If your check inherits from `CL_CI_TEST_ABAP_COMP_PROCS` or `CL_CI_TEST_SCAN`, then you can skip this section. Otherwise, your check needs to tell the framework what sort of object types it is intended to check. This is done by calling the inherited method `ADD_OBJ_TYPE` with the types of the `R3TR` objects you want to check as argument. Note that it is not possible to check `LIMU` subobjects or logical transport objects without a corresponding permanent `TADIR` entry.

Defining the messages of your check

The next important thing you need to do in the instance constructor is to create and register the messages your check will output when it encounters a finding. These messages are stored in the instance attribute `SCIMESSAGES`, and you register them simply by inserting them into this table. Here is a sample insertion of a message:

```
INSERT VALUE scimessage( test = myname
                         code = c_code_terrible_error
                         kind = c_error
                         text = 'This is a terrible mistake!'(100)
                         pcom = c_pcom_is_alright
                         pcom_alt = '' )
       INTO TABLE scimessages .
```

- `TEST` is simply the name of your check class
- `CODE` is a 10-character code that encodes the type of finding. It is highly recommended that you do not pass a literal here but instead define a reusable constant containing your code for this message. The code identifies the relevant message for a finding. It forms the link between finding and the displayed message text.
- `KIND` is the priority of your finding, encoded in a single character. You can choose between an error, a warning or simply information. These are represented by the characters `E`, `W` and `I`, respectively, but `CL_CI_TEST_ROOT` also already provides you with the predefined constants `C_ERROR`, `C_WARNING` and `C_NOTE`.
- `TEXT` is the short text that is displayed as the description of a finding. It is highly recommended to use text pool elements for the description so that the text can be translated into other languages.
- `PCOM` and `PCOM_ALT` are 20-character codes that can be used to suppress the finding via pseudo comments. The framework will first look for the occurrence of the comment in `PCOM` and then for the one in `PCOM_ALT`. The naming convention is that pseudo comment codes always start with “`CI_`” to distinguish them from the obsolete pseudo comments for e.g. SLIN or the unit test framework. Please note that the “`#EC` (that is part of the actual pseudo comment) should be excluded from the `PCOM` and `PCOM_ALT` attributes if you are inheriting from `CL_CI_TEST_ABAP_COMP_PROCS` or `CL_CI_TEST_SCAN`. If you are directly inheriting from `CL_CI_TEST_ROOT`, the framework does not automatically check for pseudo comments, even if you supply them via `INFORM( ).`

The definition of the constant passed as the `CODE` parameter deserves special attention: If you want to display documentation specific to each of these codes, you *must* name the constant the same as its content,

or the same as its content with MCODE_ prepended. For example, the constant holding the error code ERROR must be named ERROR or MCODE_ERROR.

Statically unknown T100 messages

Some checks do not know the set of all possible messages at design time, and therefore are unable to define specific messages in their constructor. For these checks, it is possible to register a generic code for the finding for which then each individual finding is associated with an individual T100 message (both its short and long text) at run time. To define such a message, you should call the method REGISTER_DYN_T100_MESSAGE:

```
METHODS register_dyn_t100_message_code FINAL
  IMPORTING p_test TYPE sci_chk
            p_code TYPE sci_errc
            p_title TYPE string
            p_kind TYPE sci_errty
            p_pcom TYPE sci_pcom OPTIONAL
            p_pragma TYPE sci_pragma OPTIONAL
            p_pseudo_remote TYPE abap_bool OPTIONAL.
```

The parameters of this method correspond exactly to the components of a SCIMESSAGE, with the exceptions that the TEXT is missing and an additional parameter P_PSEUDO_REMOTE is present. TEXT is missing because the text will only be known at run time for individual messages, and if the P_PSEUDO_REMOTE parameter is set to true, the text of the T100 message associated with a specific finding will be loaded from the checked system, not the central check system. (The name of the parameter refers to the idea that the check is "faking" being remote, and in reality just gets its results from the checked system regardless of the type of (non-)remote scenario it is being executed in.)

In order to correctly associate T100 messages with your findings, you also need to report them in a specific way, see further below.

The RUN method

The RUN method is what is called by the framework when it is told to execute your check. It is called once for each object to be checked. If you have initialization or finalization routines that should be executed only once per test run – and not per object – then you can implement these in redefinitions of the RUN_BEGIN and RUN_END methods.

Note that the framework re-uses the same instance of your check to run on multiple objects in succession. If your check carries object-specific state in its instance and class attributes, you need to reset these to their initial states either at the beginning of run or in a redefinition of the inherited method CLEAR, which is called once for every check after RUN has been called on all of them.

If your check is not meant for source code analysis, then you can skip the rest of this section, since it focuses on particular patterns to be followed when inheriting from CL_CI_TEST_ABAP_COMP_PROCS.

The start of the standard RUN method of CL_CI_TEST_ABAP_COMP_PROCS looks like this:

```
METHOD run.
  DATA:
    l_refs   TYPE t_infos.

  IF get( ) = abap_false.
    RETURN.
  ENDIF.

  analyze_start( EXPORTING p_ref_check = ref_check
                 IMPORTING p_refs      = l_refs ).
```

The GET method is supposed to initialize the data collection routines of your check and should return ABAP_FALSE if any part of this initialization failed. If you want to gather different or additional data compared to your superclass, then you should redefine the GET method, too. In that case, you should always call SUPER->GET( ) at the beginning of your redefinition. Note that if the only information your check uses is already provided by CL_CI_TEST_ABAP_COMP_PROCS, it is automatically able to be executed remotely.

ANALYZE_START is where the bulk of the check starts. The class now initiates the analysis of the individual procs. In general, most checks you want to write will need to redefine the ANALYZE_PROC method to implement their logic. You can and should assume that the code you write in ANALYZE_PROC will run at least once over every proc in the code you are checking, i.e. over the entire source code. The analysis will not proceed in any guaranteed order.

The ANALYZE_PROC method

This method is the main part of your source code analysis. Its signature is as follows:

```
METHODS analyze_proc
  IMPORTING
    p_proc TYPE cl_abap_comp_procs=>t_proc_entry
    p_from TYPE i DEFAULT 1
    p_proc_ids TYPE t_proc_ids OPTIONAL
    VALUE(p_level) TYPE i DEFAULT level
    p_params TYPE t_params OPTIONAL
  CHANGING
    p_collect TYPE t_collect.
```

- P_PROC contains a single procedure.
- P_FROM contains the number of the statement from which the analysis is supposed to begin (this is e.g. useful if you aborted the analysis of this proc earlier and now want to resume it where you left off)
- P_PROC_IDS contains a list of T_PROC_ID that you can use for various purposes; it has no dedicated purpose by default.
- P_LEVEL contains the number of levels left until the analysis is aborted, e.g. a value of 5 means that the analysis will resolve calls of methods and other reusable components up to a depth of five before aborting with an exception.
- P_PARAMS contains parameters that might have been passed to the proc, e.g. it may contain the parameters of a method call.
- P_COLLECT contains various information that persists at a level coarser than individual procs, e.g. it contains a call stack.

The main information you likely want to use in your source code analysis is contained in the procedure entry itself. The most used field of the type T_PROC_ENTRY is STMTS, which contains the table of all tokenized statements within the current proc. The statement type is one of the fundamental types you will encounter in this context:

```
BEGIN OF t_stmt,
    keyword       TYPE string,
    tokens        TYPE t_tokens,
    comments      TYPE t_comments,
    include       TYPE program,
    line          TYPE i,
    column        TYPE i,
    non_buf_db_op TYPE abap_bool,
    idx           TYPE i,
    links_origins TYPE SORTED TABLE OF i WITH UNIQUE KEY table_line,
    link_blocks   TYPE i,
END   OF t_stmt .
```

-

- KEYWORD is the keyword associated with the statement. This is often, but not always, the first token of the statement, but there are statements that do not begin with a keyword. Most relevant among these are ordinary assignments, which implicitly begin with the superfluous keyword COMPUTE, and static functional method calls that do not assign a returning parameter, which implicitly begin with the fake keyword +CALL_METHOD
- TOKENS is simply the table of actual tokens the statement consists of
- COMMENTS is the table of comments that belong to the statement
- INCLUDE is the name of the include the statement appears in
- LINE and COLUMN are the statement's position within its include

In many use cases, a large part of your ANALYZE_PROC method (or its called methods) will consist of iterating over the statements of the current proc and detecting those you are interested in by examining their keywords, e.g. via a multi-pronged "CASE <STMT>-KEYWORD" statement. If what you want to do does not fit within a few lines of code for each WHEN statement, it is highly advisable to extract such code into its own methods, since otherwise the ANALYZE_PROC method rapidly becomes cluttered and difficult to read.

Reporting a finding

CL_CI_TEST_ABAP_COMP_PROCS uses a two-step approach to findings. If you wish to report finding while not inheriting from this class, please skip to the next section. During the analysis, each finding is first registered via the ADD_INFO method. After the analysis and additional optional processing of the registered findings, the findings are reported to the end user as in all other cases.

These findings are stored in the instance attribute infos and returned as the P_REF parameter of the ANALYZE_START method. The standard implementation of run in CL_CI_TEST_ABAP_COMP_PROCS simply iterates over all findings and calls inform as described above on them, so if you want to change the information that's passed onto inform or change the way the findings are displayed, you will need to redefine the run method in your check class.

The signature of the ADD_INFO method is as follows:

```
METHODS add_info
  IMPORTING
    p_include TYPE program OPTIONAL
    p_line TYPE i OPTIONAL
    p_column TYPE i OPTIONAL
    p_kind TYPE sci_errc
    p_source TYPE csequence OPTIONAL
    p_name TYPE csequence OPTIONAL
    p_stack TYPE t_stack_entries OPTIONAL
    p_stack_of_var TYPE t_stack_entries OPTIONAL
    p_proc_pos TYPE scis_proc_pos OPTIONAL
    p_no_moves TYPE sci_no_moves OPTIONAL
    p_comments TYPE cl_abap_comp_procs=>t_comments OPTIONAL
    p_program TYPE program OPTIONAL
    p_checksum TYPE sci_crc64 OPTIONAL
    p_proc TYPE cl_abap_comp_procs=>t_proc_entry OPTIONAL
    p_stmt TYPE cl_abap_comp_procs=>t_stmt OPTIONAL
    p_stmt_supplied TYPE abap_bool DEFAULT abap_true.
```

- Once again, `P_INCLUDE`, `P_LINE` AND `P_COLUMN` indicate the position of the finding in the source code.
- `P_KIND` denotes the 10-character error code you have already used in the instance constructor.
- `P_STACK` and `P_STACK_OF_VAR` are two stacks you can attach to your finding. The default expectation is that `P_STACK` contains a call stack that leads from the tested program to the actual finding.
- `P_COMMENTS` contains a list of comments belonging to the finding. The main purpose is to store all pseudo comments related to the finding to examine them in a next step for pseudo comments that would suppress this finding.
- `P_CHECKSUM` is a checksum generated from the finding's surroundings to recognize in later runs of the same test whether the surrounding code (or even the location of the finding itself) was changed.
- `P_PROC` and `P_STMT` are again indicators of the position of the finding. The default expectation is that a finding corresponds to a single statement. If you have findings that do not relate to a specific statement and therefore do not want to pass a statement as an argument, you need to set the next parameter, `P_STMT_SUPPLIED`, to `ABAP_FALSE`.

Reporting a finding directly via `INFORM`

If you do not want to sort your findings, or if you are not implementing a subclass of `CL_CI_TEST_ABAP_COMP_PROCS` you can directly call the `INFORM` method to report a finding. Findings reported this way will show up in whatever UI the end user is using to display the result. The `INFORM` method is redefined in `CL_CI_TEST_ABAP_COMP_PROCS` and behaves slightly differently from the general case:

```
methods INFORM
  importing
    P_SUB_OBJ_TYPE type TROBJTYPE optional
    P_SUB_OBJ_NAME type SOBJ_NAME optional
    P_POSITION type INT4 optional
    P_LINE type TOKEN_ROW optional
    P_COLUMN type TOKEN_COL optional
    P_ERRCNT type SCI_ERRCNT optional
    value(P_KIND) type SYCHAR01 optional
    P_TEST type SCI_CHK
    P_CODE type SCI_ERRC
    P_SUPPRESS type SCI_PCOM optional
    P_PARAM_1 type CSEQUENCE optional
    P_PARAM_2 type CSEQUENCE optional
    P_PARAM_3 type CSEQUENCE optional
    P_PARAM_4 type CSEQUENCE optional
    P_INCLSPEC type SCI_INCLSPEC optional
    P_DETAIL type XSTRING optional
    P_CHECKSUM_1 type INT4 optional
    P_COMMENTS type T_COMMENTS optional
    P_FINDING_ORIGINS type CL_CI_SCAN=>T_ORIGIN_TAB optional .
```

- `P_SUB_OBJ_TYPE` and `P_SUB_OBJ_NAME` are the location of the finding. Usually you do not need to worry about these parameters since using the `ADD_INFO` method described below will automatically generate the correct values. If you need to set this information manually, though, you need to be rather diligent about the value of `P_SUB_OBJ_NAME`, e.g. for a method of a class this will contain the name of the automatically generated include this method appears in instead of the class name.
- `P_POSITION` is obsolete and does nothing unless you are inheriting from `CL_CI_TEST_SCAN`. If you are inheriting from `CL_CI_TEST_SCAN`, then it is the index of the statement the finding refers to in the tables of statements of the currently shared instance of `CL_CI_SCAN` (static attribute `REF_SCAN`) and not passing this parameter correctly is an error.
- `P_LINE` and `P_COLUMN` specify the position of the finding within the sub-object.
- `P_ERRCNT` is an obsolete1 parameter provided for backwards-compatibility. Don't use it.
- `P_KIND` is an obsolete parameter with which you can override the priority set for the finding's error code passed in `P_CODE`. Use different error codes for findings with different priorities instead.
- `P_TEST` is the name of the check that has raised the finding, i.e. you should only in exceptional cases pass something different than your check class's own name here.
- `P_CODE` is one of the 10-character codes you defined in the instance constructor
- `P_SUPPRESS` is an obsolete parameter to which you can pass a pseudo comment that can suppress this finding. Use the `PCOM` and `PCOM_ALT` fields of the corresponding error code's entry in `SCIMESSAGES` instead.
- `P_PARAM_<N>` are parameters for use with error codes whose messages contain placeholders of the format `&<N>`. The method will automatically substitute `&<N>` with the content of `P_PARAM_<N>` when displaying the findings.
- `P_DETAIL` is where you put any additional information not covered by other parameters. EXPORT any additional information to this raw byte string. Note that unless you also write code that reads this information at another place, e.g. in the result class discussed below, it will never be accessed by the default behavior of the framework.
- `P_CHECKSUM_1` is a checksum by which you can recognize changes to the finding's surroundings. If you want to manually compute a checksum, you should use the utility class

CL_CI_PROVIDE_CHECKSUM, but the inform method of CL_CI_TEST_ABAP_COMP_PROCS also already does this automatically for you if you do not pass a value here.

- P_COMMENTS is a table of pseudo comments which is checked against the pseudo comments defined in the SCIMESSAGES table of the test. The comments are provided in the same tabular format in the PROC_DEFS[X]-STMTS[Y]-COMMENTS. Note that due to restrictions in the parser the framework is not able to process more than one comment per line.
  Caution: If your test class is inheriting from CL_CI_TEST_SCAN, this field is obsolete and has no effect, but the position specification is used to search for pseudo comments instead in the SCAN output.
- P_FINDING_ORIGINS is a table related to the classification of the code and is automatically generated from the statement and stack of a finding reported via ADD_INFO. In detail, it is a machine-readable table of all classifications (SAP code, customer code, automatically generated code…) applying to code involved in this finding. Usually, it should contain the classifications applying to all statements involved. The classification of a statement is encoded in its LINKS_ORIGINS component, which is the index for the classification in the ORIGINS table component of the proc the statement belongs to.
- In the standard configuration, the end user will see the finding together with the message you defined for this error code in the instance constructor and can navigate to the position specified by the SUBOBJECT, LINE and COLUMN. To replace this with custom behavior, you need to implement a custom result class, see below.

Reporting a finding for statically unknown T100 messages

If you defined some message codes via the REGISTER_DYN_T100_MESSAGE_CODE method, then you should report findings for these code exclusively via the INFORM_FOR_DYN_T100 method. Its interface is almost the same as that of inform, except for the following:

There is no P_DETAIL parameter and instead there is a P_T100_KEY parameter with a structure corresponding to the message class and number of a T100 message. Pass the key for the message of the finding here.

Verification of test prerequisites

Your check may require certain prerequisites in order to be executed correctly. If these only depend on the system and not on the specific objects used, you can emit these messages not as a finding, but as a verification message by implementing the method VERIFY_TEST. If you emit a verification message with a priority higher than that of an information – i.e. a message with the attribute KIND set to CL_CI_TEST_ROOT=>C_ERROR or CL_CI_TEST_ROOT=>C_WARNING – then the inspection will terminate and no check in the current check variant will be executed at all.

If you do not wish to abort the execution of the whole inspection, but you still want to notify the user of these problems, then the ATC also has the concept of a *tool failure* finding, which will not show up in the ordinary results of an ATC run but instead in its own place. To mark a finding as a tool failure, the entry of its message code in the SCIMESSAGES table needs to have the CATEGORY attribute set to one of the following constants (all are members of CL_CI_TEST_ROOT):

Constant	Meaning
C_CAT_NOT_EXECUTABLE_AT_ALL	The check is not executable at all in the current system, no object will be checked by it.
C_CAT_SPEC_OBJ_NOT_CHECKABLE	The check is not able to check this specific object, other objects will be checked.
C_CAT_CHECK_PART_NOT_EXEC	A part of the check is not executable in the current system, but its other parts will correctly check all objects.

Activating your check

Whichever option you chose, at some point after you have created an initial implementation you will very likely want to test your check by letting it run over some test code. By default, the ATC detects your check internally but will not display it in the list of available checks when creating a new check variant. To make your check available there, start the SCI transaction and go to the *Code Inspector->Management of->Tests* menu item (or press *Shift+Alt+F5*). You will see a list of all available check classes. Find yours in this list and check the check box in front of its name, then save the selection. Your check should now be available during the definition of a new check variant, with its category and description determined by what you wrote earlier in the instance constructor.

If you have defined a new category for your check, you need to also activate it on this screen in the same way.

If you want to disable your check again, visit the same management screen and uncheck the check box again.

Allowing configurable settings for your check

When creating a check variant, the SCI transaction offers a “properties” button for checks that have the instance attribute `HAS_ATTRIBUTES` set to `ABAP_TRUE`. To not generate runtime errors in the transaction, you should only set this attribute to true after you have implemented the methods `QUERY_ATTRIBUTES`, `GET_ATTRIBUTES` and `PUT_ATTRIBUTES` belonging to the `IF_CI_TEST` interface.

The latter two methods typically just require you to export respectively import the configurable attributes of your check to/from a byte string that is passed in their sole parameter `P_ATTRIBUTES`.

The `QUERY_ATTRIBUTES` method, however, implements the dialog that is shown to the user so that they can configure the attributes manually.

As a first step, you need to define a table of type `SCI_ATTTAB` that will hold one row for each configurable attribute. Of the column, three are relevant for you: ref expects a reference to the parameter that should be configured, text is the text shown to the user in the configuration dialog and kind is a one-character flag indicating the type of the attribute. If you leave it initial, the user will see a standard text box to enter arbitrary strings for this attribute. To present a checkbox to the user, i.e. indicate a Boolean type, set the flag to ‘`C`’. To allow the user to enter a table instead of single values, set the flag to ‘`L`’. A typical statement for an attribute might look like this:

```
INSERT VALUE #( ref = REF #( level ) text = 'External analysis depth'(002)
                kind = ' ' ) INTO TABLE attributes.
```

Note that the logic that automatically generates the settings dialog from this table only works for DDIC types, i.e. the variable whose reference you pass in `REF` must have a type from the dictionary, and not a locally defined type or a type from a type group.

In this case, level could be an instance attribute of the check class that controls how deep the check descends into method, function or perform calls into other programs.

The table summarizes different options for the field `KIND`:

`KIND =`	meaning
`'G'`	horizontal line as optical separator, the value of the `TEXT` field is displayed as heading, the `REF` field has no meaning. However, any reference may be passed.
`'R'`	radio button
`'C'`	checkbox
`'S'`	select option, the type of the reference passed should be a `RANGE OF`

'L'	List box (drop down), for parameters whose domain has fixed suggested values
''	text field allowing for text input
'T'	table

After filling the attribute table with all attributes, you want the user to be able to configure in this way, you need to display the configuration dialog to the user by calling the class method CL_CI_QUERY_ATTRIBUTES=>GENERIC, which has the following signature:

```
class-methods GENERIC
  importing
    P_NAME type SCI_CHK
    P_TITLE type C
    P_ATTRIBUTES type SCI_ATTTAB
    P_MESSAGE type C optional
    P_DISPLAY type FLAG
  returning
    value(P_BREAK) type SYCHAR01 .
```

- P_NAME is the name of your check class
- P_TITLE is the title the configuration dialog should display
- P_ATTRIBUTES is your table of attributes you filled in the first step
- P_MESSAGE is a message that is displayed to the user in case their entries are erroneous
- P_DISPLAY is a parameter of QUERY_ATTRIBUTES that should simply be passed on

The return value of this function becomes ABAP_TRUE if the user cancels the dialogue. Be aware that calling this method will overwrite the contents of whatever was in the parameters referenced by the rows of P_ATTRIBUTES, so it is not recommended to directly pass references to your instance attributes unless you are fine with potentially arbitrary user input ending up in them.

Of course, if you did not pass references to the instance attributes but to local variables you need to write the content of the local variables to the instance attributes after GENERIC has finished without the user cancelling.

If the standard values of the attributes of your check after instantiation are not suitable for its execution, then you should set the ATTRIBUTES_OK variable to ABAP_FALSE in its constructor.

RESULTS AND RESULT CLASSES

If you want to output your results in a form different from the standard output, you will need to implement this output logic in its own dedicated result class inheriting from CL_CI_RESULT_ROOT. The naming convention is that the result class belong to a check class CL_CI_TEST_MYTEST is called CL_CI_RESULT_MYTEST, however this is not enforced. You need to register the result class in your test class by redefining the method GET_RESULT_NODE to return an instance of your new custom result class. This method possesses an importing parameter P_KIND that you should simply pass to the constructor of your result class, and from there to the constructor of its superclass. However, none of this will be necessary unless you are dissatisfied with the way your results are displayed by the default logic.

Short texts and long texts

The short and long texts that are displayed for a finding are determined by the GET_TEXT and GET_DOCU_FOR_TEST_CODE methods of the result class. By implementing these methods, you can exact detailed control over how the results of your check are displayed.

Navigation and stacks

If you passed a valid source code position when reporting your finding via `INFORM`, then navigation to the location of the finding works automatically in both SAPGUI and ADT. However, sometimes you might want to enable the user to navigate to additional locations in the source code – the archetypal example is when your finding is related to the structure of a call stack, and you want to display the call stack and allow the user to navigate to the source code position of the individual entries.

To achieve this, you should export a table of type `SCIT_WB_NAVIGATION` under the name `WB_NAVIGATION` to the details buffer of the finding. The row type `SCIS_WB_NAVIGATION` has the following components:

```
define structure scis_wb_navigation {
  object_type      : sci_wb_object_type;
  object_name      : sci_wb_object_name;
  enclosing_object : sci_wb_enclosing_object;
  description      : sci_wb_description;
  position         : sci_wb_position;
}
```

`POSITION` is a line number in the source code-like object given by the triple `OBJECT_TYPE`, `OBJECT_NAME`, `ENCLOSING_OBJECT`. In this case, `OBJECT_TYPE` and `OBJECT_NAME` refer to the include, and `ENCLOSING_OBJECT` to the `TADIR` object the include belongs to. `DESCRIPTION` is the text that will be displayed as the text of the link that leads to the position specified by the other components.

Alternatively, you can override the entire logic of creating the workbench navigation by implementing the method `GET_WB_NAVIGATION` for your result class.

Caution: Result classes also have an `IF_CI_TEST~NAVIGATE` method. This method is a legacy functionality that only steers navigation in the SCI transaction, it has no effect on navigation in the ATC.

UNIT TESTS FOR YOUR CHECK

Testing the functionality of your check manually at each step of the development process is time consuming and prone to errors. Fortunately, the ABAP Unit Test framework together with the class `CL_CI_TEST_VERIFY` provide a convenient way to automate the testing process. Your unit test classes should therefore inherit from this class.

Running your check during a unit test

`CL_CI_TEST_VERIFY` offers a run method that will carry out your check on a single ABAP dictionary object or program. Its signature is given below, but regardless of your specific use case you need to have created a *global* check variant that runs only your test and nothing else.

```
class-methods RUN
  importing
    P_VARIANT type SCI_CHKV
    P_OBJ_TYPE type TROBJTYPE
    P_OBJ_NAME type SOBJ_NAME
    P_OBJ_PARAMS type SCIT_OBJ_PAR optional
    P_NOSUPPRESS type SCI_NOSUP optional
    P_SYSID type SYST_SYSID optional
    P_DESTINATION type RFCDEST optional
    P_ALLOW_EXCEPTIONS type ABAP_BOOL default ABAP_FALSE
  raising
    CX_CI_CHECK_ERROR .
```

- `P_VARIANT` is the name of your global check variant as a string
- `P_OBJ_TYPE` and `P_OBJ_NAME` are the `TADIR` keys of the object you wish to check

- P_OBJ_PARAMS is an optional parameter in which you can pass the parameters the framework would normally determine by itself during an ordinary inspection, such as if the object to be checked is an SAP object, or which namespace it belongs to. For the kinds of allowed parameters refer to the definition of C_OBJ_PARAM_KINDS in CL_CI_OBJECTSET.
- P_NOSUPPRESS is a flag to deactivate the suppression of findings via pseudo comments for this check run.
- P_SYSID is the RFC source in case you want to test RFC functionality. The same caveats as for P_DESTINATION below apply.
- P_DESTINATION is the RFC destination in case you want to test RFC functionality. This is generally advised against for elementary unit testing since you make yourself dependent on the remote availability of the destination but may be useful in some cases. If you do want to include remote executions of your check in your unit tests, consider making their failure tolerable so that the unavailability of the remote system does not prevent the execution of other tests and is not seen as a fatal error.
- P_ALLOW_EXCEPTIONS determines if the exceptions for a missing check variant or a missing object to be checked cause a failure of the unit test (this happens if this parameter is ABAP_FALSE) or the corresponding exceptions are merely propagated by the method.

Usually you only need to pass the required parameters to do useful unit testing. After this method has been called, the results of the check will be stored in the instance attribute RESULT_LIST of your check class.

Checking your findings against your expectations

The simplest check is of course to verify that you have the expected number of findings, i.e. that LINES(RESULT_LIST) has the expected value, but generally you will want to ensure all findings are also raised with the correct error codes and positions. CL_CI_TEST_VERIFY provides the check method for this purpose.

```
class-methods CHECK
  importing
    P_CODE type SCI_ERRC
    P_SOBJTYPE type TROBJTYPE
    P_SOBJNNAME type SOBJ_NAME
    P_LINE type I
    P_COL type I
    P_PARAM_1 type STRING optional
    P_PARAM_2 type STRING optional
    P_PARAM_3 type STRING optional
    P_PARAM_4 type STRING optional
    P_DETAIL type SCIT_DETAIL optional
    P_CHECKSUM1 type I optional
    P_TEST type SCI_CHK optional
  returning
    value(P_MESSAGE) type STRING .
```

You will certainly note that, not coincidentally, these parameters are the same as those for the INFORM method. Simply pass the values you expect for your findings to this method. If the actual finding matches the expected finding, the returned P_MESSAGE string will be empty, otherwise it will contain information about the mismatch, usually the data of the finding that was expected.

Pay attention to the unusually spelled parameter P_SOBJNNAME – preserved this way for backward-compatibility – as well as once again to the fact that this SUBOBJECT identifier refers to the full name of the actual include, not e.g. the class, a finding was reported in.

CREATING DOCUMENTATION

You can create documentation that will automatically be displayed in both the SCI transaction and the ATC, regardless of whether it is being used through SAPGUI or the ADT. There are two basic types of documentation, the documentation of the check itself and the documentation of each error code of a check. Note that you need to set the attribute `HAS_DOCUMENTATION` to `ABAP_TRUE` for any documentation to be displayed.

The documentation of the check itself is always created in the SE61 transaction by documenting the dummy class attribute `0000` of your check class. This documentation will appear when a user clicks on the blue information icon in the SCI transaction next to your check and will also be displayed by the ATC for each finding of your check.

You can furthermore document each error code of your check by documenting class attributes in SE61 whose names agree with the literal values of your error code constants, not with these constants' names. Alternatively, you can document an attribute named `MCODE_<LITERAL>`. Note that this means that your error code constants must be named exactly like their values, or like their values with `MCODE_` prepended, otherwise SE61 will not allow you to create documentation – you can't document class attributes that do not exist. This documentation – if necessary – should contain e.g. the reason this finding is being displayed and advice on how to fix this specific finding.

You should consider the documentation of the check itself mandatory, since otherwise only you will be able to know what the check even is supposed to do. In contrast, documentation for individual error codes should only be considered in cases where the short text of the finding is not self-explanatory or where the course of action to fix the finding is not obvious. You do not need to explicitly document any pseudo comments that suppress a finding; the framework will automatically display up to two pseudo comments that can suppress the finding in its description.

Custom documentation access

You are not restricted to the behavior for defining and retrieving documentation outlined above. The result class of your check steers how documentation for specific findings is retrieved, the above is just the default behavior. If you want to display documentation in a different way, you should implement the method `GET_DOCU_FOR_TEST_CODE` in your result class.

ENSURING REMOTE ABILITY OF YOUR CHECK

ATC checks can analyze development objects on other SAP systems in the same landscape. The source code information – and any other local information a check needs – needs to be retrieved from the target system via RFC.

There is a series of blog entries on how to perform remote analysis via ATC once you have a remote-enabled check.

For your check to be able to run successfully in a remote scenario, you need to avoid any explicit dependence on local data of the check system. For instance, a common pitfall is to rely on reading the `TADIR` or other tables with object information – of course that will only get you information about the objects in your check system, not about the objects in the checked system. You should also be wary of using kernel methods to provide information or using the RTTI (**R**un **T**ime **T**ype **I**nformation) functionality.

If you use only the information and methods provided by `CL_CI_TEST_ABAP_COMP_PROCS`, you can be sure that all relevant data is acquired from the checked system. However, if you require additional data that is not yet provided by this class, you currently need to write your own RFC function module that will make the necessary data available to your check class.

Of course, when you manually call an RFC module in your test, you need to ensure that the destination is correct. The destination is not directly passed to your check, but in an enriched format called the `SCR_SOURCE_ID`. The currently valid source ID is held globally in the class attribute `CL_CI_TEST_ROOT=>SRCID`. To translate this into an RFC destination, you should call the method `CL_ABAP_SOURCE_ID=>GET_DESTINATION`.

CREATING QUICKFIXES

The following section is only relevant if your check reports findings in ABAP source code.

In many cases, findings have an easy and obvious solution, such as appending a particular addition to an ABAP statement, deleting an unused variable or other minor modifications of the source code. In cases where it is possible to statically determine the correct solution to the problem reported by the finding, it is natural to want to offer this solution programmatically to the user.

Quickfixes in the ABAP Development Tools (ADT) make this possible: You can attach arbitrarily many so-called quickfixes to a finding, which can then be executed by the developer in the IDE. For a set of findings, it is even possible to execute a quickfix for each finding at once, vastly reducing the time spent by developers compared to manually applying the same solution to each finding individually.

Creating a set of quickfixes

A finding can have an arbitrary number of quickfixes attached to it. The container for these fixes is an instance of `CL_CI_QUICKFIX_CREATION`, which you should instantiate by calling the method `CL_CI_QUICKFIX_CREATION=>CREATE_QUICKFIX_ALTERNATIVES`.

This container now allows you to create individual quickfixes by calling `CREATE_QUICKFIX`. If you create more than one potential fix for a single finding and you want to make it possible to identify which fix is which later on, you should pass the optional parameter `P_QUICKFIX_CODE` with a character sequence of your choosing that identifies the type of fix being offered.

Defining the actions of a quickfix

Currently, only quickfixes acting on ABAP code are supported, but this may change in the future, so the type of a quickfix (`IF_CI_QUICKFIX_SINGLE`) does not directly offer ABAP actions, but instead merely includes the interface `IF_CI_QUICKFIX_ABAP_ACTIONS`. The currently supported actions are

- `ADD_PSEUDO_COMMENT`
- `INSERT_AFTER`
- `INSERT_BEFORE`
- `REPLACE_BY`

All actions have in common that they rely on the notion of a context, which needs to be passed as the `P_CONTEXT` parameter of the actions.

While you do not need to worry about the technical underpinnings of this notion, you should think of a context as something that uniquely identifies the place in an ABAP program where the quickfix should take place. So regardless of which of the above actions you wish to perform, you must first create a context that specifies where this action should be performed. Contexts can be created from a line/column specification in an include, a collection of `SCAN` tokens or statements, or a collection of `COMP_PROCS` tokens or statements via the static factory methods in `CL_CI_QUICKFIX_ABAP_CONTEXT`, starting with `CREATE_FROM_SCAN_` and `CREATE_FROM_COMP_PROCS_`, respectively.

This context determines where the action you specify is performed. `ADD_PSEUDO_COMMENT` adds the comment at a valid position for every statement in the context, `INSERT_AFTER` inserts code at the end of the context, `INSERT_BEFORE` inserts code at the beginning of the context and `REPLACE_BY` replaces the whole context by entirely new code.

Since you can create contexts both from statements and from tokens, you are therefore able to delete, insert or replace entire statements as well as individual tokens. The example class `CL_CI_TEST_DOCU_EXAMPLE` demonstrates this: The quickfix that transforms a `MOVE` into an assignment via = acts on the scope of the whole statement and creates its context as

```
p_context  = cl_ci_quickfix_abap_context=>create_from_comp_procs_stmt(
  p_action_stmt    = statement
  p_proc_def       = procedure
```

```
      p_proc_defs      = proc_defs
      p_with_end_point = abap_false
    )
```

while the quickfix that deletes the `COMPUTE` keyword acts on the scope of the `COMPUTE` token and creates its context as

```
  p_context  = cl_ci_quickfix_abap_context=>create_from_comp_procs_tokens(
    p_action_stmt = statement
    p_proc_def = procedure
    p_proc_defs = proc_defs
    p_from_token = 1
  )
```

so that the context is only the first token of the `COMPUTE` statement.

Quickfixes can be enabled for automatic execution, so that the ADT will apply them when the use tells its to perform all recommended quickfixes. To do so, simply call the method `ENABLE_AUTOMATIC_EXECUTION` on the quickfix object.

Defining the display text of a quickfix

For the fix to be displayed correctly to the user, you must associate short and long texts to the quickfix that can be showing in the dialog where the user chooses which of the quickfixes attached to a finding to apply.

To attach the short and long text of a message class to a quickfix, call the `ADD_DOCU_FROM_MSGCLASS` method on it. The screenshot below shows where these texts then appear in the IDE.

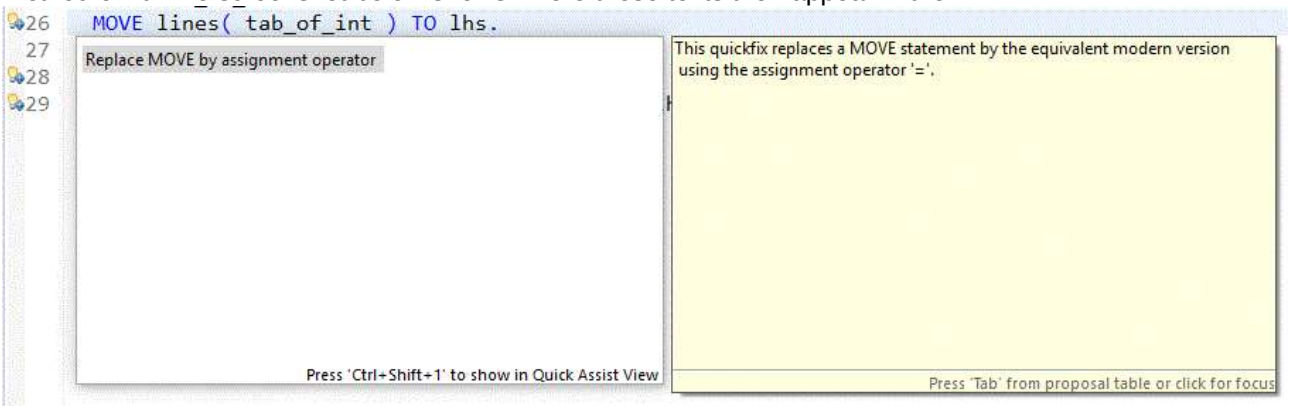

Figure 2 The texts associated with a quickfix. Left: Short text. Right: Long text.

Unit testing your quickfix

Since writing quickfixes that always produce the intended (and syntactically correct) code can be a subtle matter, you can add functionality to your unit tests that ensures that the fix attached to a finding is both syntactically correct and produces the exact code you expect (ignoring non-significant whitespace).

Quickfixes are tested via an instance of `CL_CI_QUICKFIX_TESTING`. You set the finding whose quickfix you want to test by calling `SET`, and you start the check via either `CHECK_QUICKFIX` or `CHECK_QUICKFIXES` methods. There are two independent parts to this check: First, you can execute a syntax check for the result of the quickfix. This option is on by default and is recommended in almost all cases to detect quickfixes that produce incorrect code. In rare situations it may be unavoidable to produce errors, but in most cases you should ensure that your quickfix never produces syntax errors. Second, you can test whether the content of the fix is the code you expect by passing the `P_EXPECTED_QUICKFIXES` parameter of `CHECK_QUICKFIXES`.

AN EXAMPLE: DETECTING OBSOLETE KEYWORDS

As an illustration of how to use the functionality outlined above, the following chapter describes the implementation of a specific check in detail.

Requirements

The new check is aimed at modernizing old code. It should detect `MOVE` and `COMPUTE` statements and build a quickfix that replaces them with their non-obsolete form, i.e. a keyword-less assignment using the = operator. We start with an empty class `CL_CI_TEST_DOCU_EXAMPLE` inheriting from `CL_CI_TEST_ABAP_COMP_PROCS` since we want to analyze ABAP source code.

Defining the check meta data

The meta data of the check are defined in its constructor as follows:

```
METHOD constructor.
  super->constructor( ).
  description = 'Example: Find MOVEs and COMPUTEs'(des).
  category    = 'CL_CI_CATEGORY_INTERNAL'.
  position    = '005'.
  remote_rfc_enabled  = abap_true.
  check_scope_enabled = abap_false.
  has_attributes      = abap_false.
  INSERT LINES OF VALUE scimessages(
    ( test = my_name code = finding_codes-simple_move kind = c_note text = 'The MOVE
statement is obsolete'(mov) )
    ( test = my_name code = finding_codes-compute kind = c_note text = 'The COMPUTE
keyword is obsolete'(cmp) )
  ) INTO TABLE scimessages.

ENDMETHOD.
```

Since we only intend to use the source code information provided by `CL_CI_TEST_ABAP_COMP_PROCS`, the check is `REMOTE_RFC_ENABLED`, and since we do not intend to provide any special handling of generated or SAP code, it is not `CHECK_SCOPE_ENABLED`. We also define two finding codes for the check – one for `MOVE` statements and the other for `COMPUTE` statements. Since these statements are not functional errors, but merely cosmetic annoyances, we give the findings the lowest possible priority as `C_NOTE`. Note also that all texts that will be displayed to the user are text elements, so that they can be translated.

Test cases

Before implementing the check logic, we want to have some simple test cases for us to test the check against. We create the reports `RS_CI_TEST_DOCU_EXAMPLE_1` and `RS_CI_TEST_DOCU_EXAMPLE_2` containing some `MOVE` and `COMPUTE` statements respectively.

After activating our new check class in the SCI transaction via the menu item *Code Inspector->Management of->Checks (Ctrl+Shift+F5)*, we define a global check variant containing only this check called `DOCU_EXAMPLE`. Of course, running this variant gives us no findings yet since we haven't implemented any logic in our check.

The check logic

We start by redefining the method `ANALYZE_PROC` inherited from the superclass. The core logic of our check is rather simple and can be expressed in a single short loop:

```
  LOOP AT p_proc-stmts ASSIGNING FIELD-SYMBOL(<statement>)
      WHERE keyword = 'MOVE' OR keyword = 'MOVE-CORRESPONDING' OR keyword = 'COMPUTE'.
    CASE <statement>-keyword.
      WHEN 'MOVE' OR 'MOVE-CORRESPONDING'.
        emit_finding(
          procedure = p_proc
          statement = <statement>
          code = finding_codes-simple_move
        ).

      WHEN 'COMPUTE'.
        ASSIGN <statement>-tokens[ 1 ] TO FIELD-SYMBOL(<first_token>).
        IF <first_token>-refs IS INITIAL AND <first_token>-str = 'COMPUTE'.
          emit_finding(
            procedure = p_proc
            statement = <statement>
            code = finding_codes-compute
          ).
        ENDIF.
    ENDCASE.
  ENDLOOP.
```

`EMIT_FINDING` is just a wrapper around `CL_CI_TEST_ABAP_COMP_PROCS->INFORM` that causes a finding to the reported to the check framework. We simply loop over all statements and report one finding for every statement that begins with the keywords `MOVE` or `COMPUTE`. If we didn't want to build quickfixes for these statements, we would already be done now.

Building quickfixes

We define the `BUILD_MOVE_QUICKFIXES` and `BUILD_COMPUTE_QUICKFIXES` methods that take the current `PROCEDURE` and `STATEMENT` and return an instance of `CL_CI_QUICKFIX_CREATION`, the container type for a set of quickfixes. The essence of BUILD_MOVE_QUICKFIXES is the following logic:

```
METHOD add_move_to_assignment_op_fix.
  DATA(fix) = CAST if_ci_quickfix_abap_actions( quickfixes->create_quickfix( ) ).
  DATA(move_info) = parse_move( statement ).
  fix->replace_by(
    p_new_code_tab = CONV #( construct_equiv_assignment( move_info ) )
    p_context  = cl_ci_quickfix_abap_context=>create_from_comp_procs_stmt(
      p_action_stmt    = statement
      p_proc_def       = procedure
      p_proc_defs      = proc_defs
      p_with_end_point = abap_false
    )
  ).
  CAST if_ci_quickfix_single( fix )->add_docu_from_msgclass(
    p_msg_class  = quickfix_docu_message_class
    p_msg_number = 001
  ).
  CAST if_ci_quickfix_single( fix )->enable_automatic_execution( ).

ENDMETHOD.
```

`PARSE_MOVE` obtains relevant info about the statement by detecting whether it contains `EXACT` or is a `MOVE-CORRESPONDING` or whether it is a cast of object references (i.e. contains `?TO` instead of `TO`). It also separates the statement into its `SOURCE` and `TARGET`, according to `MOVE SOURCE TO TARGET`.

The actual quickfix now consists of two parts: the code that shall replace the `MOVE` statement, which we construct in `CONSTRUCT_EQUIV_ASSIGNMENT` and the context that specifies which part of the code we want to replace. Since we want to replace the entire statement, we create the context directly from the statement.

See the appendix for the implementation of methods like `PARSE_MOVE` or `CONSTRUCT_EQUIV_ASSIGNMENT` if you are interested.

We are confident that this quickfix cannot break anything, so we call `ENABLE_AUTOMATIC_EXECUTION` on it, meaning that users can carry out this quickfix in bulk. We also define a short and long text in a message class, leading to the texts in Figure 2 above being shown to the user.

We finally pass the result of `ADD_MOVE_TO_ASSIGNMENT_OP_FIX` to `EMIT_FINDING`, where the quickfix is transported as the detail of the finding by passing it as

```
p_detail = COND #( WHEN quickfix IS NOT INITIAL THEN quickfix->export_to_xstring( ) )
```

to the `INFORM` method. This finishes the fix for the `MOVE` statement.

The second fix is very easy at first sight: When we see a compute, we delete it and leave the rest of the statement untouched:

```
METHOD add_delete_compute_fix.
  DATA(fix) = CAST if_ci_quickfix_abap_actions( quickfixes->create_quickfix( ) ).
  fix->replace_by(
    p_new_code = ``
    p_context  = cl_ci_quickfix_abap_context=>create_from_comp_procs_tokens(
      p_action_stmt = statement
      p_proc_def = procedure
      p_proc_defs = proc_defs
      p_from_token = 1
    )

  ).
```

Since there is no explicit delete action, deletion of tokens is represented by an empty replace action on the context that just consists of the first token of the statement.

For simple `COMPUTE` statements, this is already the entire fix. However, there is a variant of the compute statement with the second token being an `EXACT` keyword, where `COMPUTE EXACT A = B` is equivalent to the statement `A = EXACT #( B )`. So if there is an `EXACT`, we need to delete the `EXACT` and surround the right hand side of the assignment with the `EXACT` operator:

```
  ASSIGN statement->tokens[ 2 ] TO FIELD-SYMBOL(<second_token>).
  IF <second_token>->refs IS INITIAL AND <second_token>->str = 'EXACT'.
    fix->replace_by(
      p_new_code = ``
      p_context = cl_ci_quickfix_abap_context=>create_from_comp_procs_tokens(
        p_action_stmt = statement
        p_proc_def = procedure
        p_proc_defs = proc_defs
        p_from_token = 2
      )
    ).
    DATA(rhs_context) = cl_ci_quickfix_abap_context=>create_from_comp_procs_tokens(
      p_proc_defs = proc_defs
      p_proc_def = procedure
      p_action_stmt = statement
      p_from_token = 5
      p_to_token = lines( statement->tokens )
    ).
    fix->insert_before(
      p_new_code = `EXACT #(`
      p_context = rhs_context
    ).
```

```
    fix->insert_after(
      p_new_code = `)`
      p_context = rhs_context
    ).
  ENDIF.
```

Since we can create fine-grained contexts on the level of individual tokens, we represent our fix by three distinct actions: Deleting the `EXACT`, inserting the token sequence `EXACT #(` in front of the right hand side and inserting the token `)` behind the right hand side. The insertions are performed by the quickfix actions `INSERT_BEFORE` and `INSERT_AFTER`, and we can always let the context start at the fifth token of the statement because `COMPUTE` statements only allow single tokens as the left hand side of the assignment (i.e. no table expressions). This completes the quickfix for `COMPUTE` statements.

Unit tests

At some point during the development of our check, we of course want to add unit tests to it (this being the last section in this example is not supposed to be a suggestion to only add tests at the end). Since the signatures of the generic unit test methods provided by the framework are rather unwieldy for our specific use case, we define a custom assertion method in our test class inheriting from `CL_CI_VERIFY_TEST`:

```
METHODS assert_finding
  IMPORTING finding_code TYPE sci_errc
            position TYPE cl_ci_test_docu_example=>ty_position
            expected_quickfixed_code TYPE ty_expected_quickfix
  RAISING cx_static_check.

METHOD assert_finding.
  DATA(error_message) = check(
    p_test      = cut_name
    p_code      = finding_code
    p_sobjtype  = 'PROG'
    p_sobjname = position-include
    p_line      = position-line
    p_col       = position-column
  ).
  IF error_message IS NOT INITIAL.
    cl_abap_unit_assert=>fail(
      msg = error_message
      quit = if_abap_unit_constant=>quit-no
    ).
  ELSE.
    DATA(quickfix_tester) = NEW cl_ci_quickfix_testing( ).
    quickfix_tester->set( p_detail = result_list[
       sobjtype = 'PROG'
       sobjname = position-include
       line = CONV #( position-line )
       col = CONV #( position-column )
     ]-detail ).
    DATA(errors) = VALUE cl_ci_quickfix_testing=>t_err_msgs( ).
    DATA(quickfixed_sources) = VALUE cl_ci_quickfix_testing=>t_result_sources( ).
    DATA(quickfix_okay) = quickfix_tester->check_quickfixes(
      EXPORTING
        p_program = position-include
        p_expected_quickfixes = VALUE #( (
          qf_code      = 'QFIX1'
```

```
          replacements = VALUE #( (
            include   = position-include
            from_line = position-line
            to_line   = position-line + lines( expected_quickfixed_code ) - 1
            by        = CONV #( expected_quickfixed_code )
          ) )
        ) )
      IMPORTING
        p_err_msgs = errors
        p_result_sources = quickfixed_sources
    ).
    IF quickfix_okay = abap_false.
      DATA(first_error) = errors[ 1 ].
      cl_abap_unit_assert=>assert_true(
        act = quickfix_okay
        msg = |Quickfix error: { first_error-msg }. Erroneous line: {
quickfixed_sources[ include = first_error-include ]-source[ first_error-line ] }|
        quit = if_abap_unit_constant=>quit-no
      ).
    ENDIF.
  ENDIF.

ENDMETHOD.
```

This method first checks whether there exists a finding with the intended code and location at all in the check result. If it does, then we instantiate the quickfix test tool, pass the finding to it via `SET` and then construct the correct expected replacement from the expected raw code we take as input. If the test tool reports an error, we show the first error message and the corresponding line in the code with the quickfix applied to it.

While this method may look a bit unwieldy, it makes for readable assertions in our actual test methods: After running our check against our test objects as

```
  DATA(test_report) = CONV sobj_name( 'RS_CI_TEST_DOCU_EXAMPLE_1' ).
  run(
    p_variant  = 'DOCU_EXAMPLE'
    p_obj_type = 'PROG'
    p_obj_name = test_report
    p_create_quickfixes = abap_true
  ).
```

we can write an assertion for a finding including a quickfix as

```
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 23 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `lhs = rhs.` )
    )
  ).
```

which corresponds to a line `MOVE RHS TO LHS.` in our test code. After writing an assertion for every expected finding, we can quickly check whether our check produces the quickfixes we expect and that they produce syntactically correct ABAP code when applied just by running the unit tests.

APPENDIX

Full example code

Here's the complete class `CL_CI_TEST_DOCU_EXAMPLE` and its two test reports:

Class CL_CI_TEST_DOCU_EXAMPLE

```
CLASS cl_ci_test_docu_example DEFINITION
  PUBLIC
  FINAL
  INHERITING FROM cl_ci_test_abap_comp_procs
  CREATE PUBLIC .

PUBLIC SECTION.
  TYPES:
    BEGIN OF ty_position,
      include TYPE program,
      line TYPE i,
      column TYPE i,
    END OF ty_position.

  CONSTANTS:
    BEGIN OF finding_codes,
      simple_move TYPE sci_errc VALUE 'MOVE',
      compute TYPE sci_errc VALUE 'COMPUTE',
    END OF finding_codes.

  METHODS constructor.
PROTECTED SECTION.
  METHODS analyze_proc REDEFINITION.
PRIVATE SECTION.
  TYPES ty_procedure TYPE cl_abap_comp_procs=>t_proc_entry.
  TYPES ty_tokens TYPE cl_abap_comp_procs=>t_tokens.
  TYPES ty_statement TYPE cl_abap_comp_procs=>t_stmt.
  TYPES ty_quickfixes TYPE REF TO cl_ci_quickfix_creation.
  TYPES:
    BEGIN OF ty_move_info,
      source TYPE ty_tokens,
      target TYPE ty_tokens,
      is_exact TYPE abap_bool,
      is_corresponding TYPE abap_bool,
      is_cast TYPE abap_bool,
    END OF ty_move_info.
  TYPES ty_source_code TYPE STANDARD TABLE OF string WITH EMPTY KEY.

  CONSTANTS my_name TYPE sci_chk VALUE 'CL_CI_TEST_DOCU_EXAMPLE'.
  constants quickfix_docu_message_class type syst_msgid value 'CI_DOCU_EXAMPLE'.

  DATA analyzed_proc_ids TYPE HASHED TABLE OF cl_abap_comp_procs=>t_proc_id WITH UNIQUE
KEY table_line.

  METHODS emit_finding
    IMPORTING procedure TYPE ty_procedure
              statement TYPE ty_statement
              code TYPE sci_errc
```

```
              quickfixes TYPE ty_quickfixes.
  METHODS build_move_quickfixes
    IMPORTING procedure TYPE ty_procedure
              statement TYPE ty_statement
    RETURNING VALUE(quickfixes) TYPE ty_quickfixes.
  METHODS build_compute_quickfixes
    IMPORTING procedure TYPE ty_procedure
              statement TYPE ty_statement
    RETURNING VALUE(quickfixes) TYPE ty_quickfixes.
  METHODS add_move_to_assignment_op_fix
    IMPORTING procedure TYPE ty_procedure
              statement TYPE ty_statement
    CHANGING quickfixes TYPE ty_quickfixes.
  METHODS parse_move
    IMPORTING statement TYPE ty_statement
    RETURNING VALUE(move_info) TYPE ty_move_info.
  METHODS flatten_tokens
    IMPORTING tokens TYPE ty_tokens
    RETURNING VALUE(code) TYPE string.
  METHODS compute_statement_checksum
    IMPORTING procedure TYPE ty_procedure
              statement TYPE ty_statement
    RETURNING VALUE(checksum) TYPE sci_crc64.
  METHODS construct_equiv_assignment
    IMPORTING move_info TYPE ty_move_info
    RETURNING VALUE(new_statement) TYPE ty_source_code.
  METHODS break_into_lines
    IMPORTING code TYPE string
    RETURNING VALUE(code_lines) TYPE ty_source_code.
  METHODS add_delete_compute_fix
    IMPORTING procedure TYPE ty_procedure
              statement TYPE ty_statement
    CHANGING  quickfixes TYPE ty_quickfixes.
ENDCLASS.


CLASS cl_ci_test_docu_example IMPLEMENTATION.

METHOD constructor.
  super->constructor( ).
  description = 'Example: Find MOVEs and COMPUTEs'(des).
  category    = 'CL_CI_CATEGORY_INTERNAL'.
  position    = '005'.
  remote_rfc_enabled  = abap_true.
  check_scope_enabled = abap_false.
  has_attributes      = abap_false.
  INSERT LINES OF VALUE scimessages(
    ( test = my_name code = finding_codes-simple_move kind = c_note text = 'The MOVE
statement is obsolete'(mov) )
    ( test = my_name code = finding_codes-compute kind = c_note text = 'The COMPUTE
keyword is obsolete'(cmp) )
  ) INTO TABLE scimessages.
ENDMETHOD.

METHOD analyze_proc.
  IF line_exists( analyzed_proc_ids[ table_line = p_proc-proc_id ] ).
    RETURN.
```

```
  ELSE.
    INSERT p_proc-proc_id INTO TABLE analyzed_proc_ids.
  ENDIF.
  LOOP AT p_proc-stmts ASSIGNING FIELD-SYMBOL(<statement>)
      WHERE keyword = 'MOVE' OR keyword = 'MOVE-CORRESPONDING' OR keyword = 'COMPUTE'.
    CASE <statement>-keyword.
      WHEN 'MOVE' OR 'MOVE-CORRESPONDING'.
        emit_finding(
          procedure = p_proc
          statement = <statement>
          code = finding_codes-simple_move
          quickfixes = COND #( WHEN is_quickfix_creation_enabled( ) THEN
build_move_quickfixes(
            procedure = p_proc
            statement = <statement>
          ) )
        ).

      WHEN 'COMPUTE'.
        ASSIGN <statement>-tokens[ 1 ] TO FIELD-SYMBOL(<first_token>).
        IF <first_token>-refs IS INITIAL AND <first_token>-str = 'COMPUTE'.
          emit_finding(
            procedure = p_proc
            statement = <statement>
            code = finding_codes-compute
            quickfixes = COND #( WHEN is_quickfix_creation_enabled( ) THEN
build_compute_quickfixes(
              procedure = p_proc
              statement = <statement>
            ) )
          ).
        ENDIF.
    ENDCASE.
  ENDLOOP.
ENDMETHOD.

METHOD emit_finding.
  inform(
    p_test         = my_name
    p_code         = code
    p_sub_obj_type = 'PROG'
    p_sub_obj_name = statement-include
    p_line         = statement-line
    p_column       = CONV #( statement-column )
    p_checksum_1   = compute_statement_checksum(
      procedure = procedure
      statement = statement
    )-i1
    p_detail = COND #( WHEN quickfixes IS NOT INITIAL THEN quickfixes-
>export_to_xstring( ) )
  ).
ENDMETHOD.

METHOD build_move_quickfixes.
  quickfixes = cl_ci_quickfix_creation=>create_quickfix_alternatives( ).
  add_move_to_assignment_op_fix(
    EXPORTING procedure = procedure
```

```
              statement = statement
    CHANGING quickfixes = quickfixes
  ).
ENDMETHOD.

METHOD add_move_to_assignment_op_fix.
  DATA(fix) = CAST if_ci_quickfix_abap_actions( quickfixes->create_quickfix( ) ).
  DATA(move_info) = parse_move( statement ).
  fix->replace_by(
    p_new_code_tab = CONV #( construct_equiv_assignment( move_info ) )
    p_context  = cl_ci_quickfix_abap_context=>create_from_comp_procs_stmt(
      p_action_stmt    = statement
      p_proc_def       = procedure
      p_proc_defs      = proc_defs
      p_with_end_point = abap_false
    )
  ).
  CAST if_ci_quickfix_single( fix )->add_docu_from_msgclass(
    p_msg_class  = quickfix_docu_message_class
    p_msg_number = 001
  ).
  CAST if_ci_quickfix_single( fix )->enable_automatic_execution( ).
ENDMETHOD.

METHOD compute_statement_checksum.
  get_stmt_checksum(
    EXPORTING p_proc_def   = procedure
              p_index_stmt = statement-idx
    CHANGING  p_checksum   = checksum
  ).
ENDMETHOD.

METHOD flatten_tokens.
  code = REDUCE #( INIT str = `` FOR tok IN tokens NEXT str = |{ str }{ tok-str } | ).
ENDMETHOD.

METHOD parse_move.
  move_info-is_corresponding = xsdbool( statement-keyword = 'MOVE-CORRESPONDING' ).
  move_info-is_exact         = xsdbool( statement-tokens[ 2 ]-str = 'EXACT' AND
statement-tokens[ 2 ]-refs IS INITIAL ).
  DATA(start_of_source) = COND #( WHEN move_info-is_exact = abap_true THEN 3 ELSE 2 ).
  DATA(found_to) = abap_false.
  LOOP AT statement-tokens FROM start_of_source ASSIGNING FIELD-SYMBOL(<token>).
    IF <token>-refs IS INITIAL AND <token>-str = 'TO'.
      found_to = abap_true.
    ELSEIF <token>-refs IS INITIAL AND <token>-str = '?TO'.
      move_info-is_cast = abap_true.
      found_to = abap_true.
    ELSEIF found_to = abap_false.
      INSERT <token> INTO TABLE move_info-source.
    ELSE.
      INSERT <token> INTO TABLE move_info-target.
    ENDIF.
  ENDLOOP.
ENDMETHOD.

METHOD construct_equiv_assignment.
```

```
  DATA(lhs) = flatten_tokens( move_info-target ).
  DATA(rhs) = flatten_tokens( move_info-source ).
  DATA(operator) = COND #(
    WHEN move_info-is_corresponding = abap_true THEN `CORRESPONDING`
    WHEN move_info-is_exact = abap_true THEN `EXACT`
    WHEN move_info-is_cast = abap_true THEN `CAST`
  ).
  DATA(needs_inner_exact) = xsdbool( move_info-is_corresponding = abap_true AND
move_info-is_exact = abap_true ).
  DATA(flat_new_statement) = COND #(
    WHEN operator IS INITIAL
      THEN |{ lhs } = { rhs }|
      ELSE |{ lhs } = { operator } #( { COND #( WHEN move_info-is_corresponding =
abap_true THEN |BASE ( { lhs } ) | ) }| &&
                                      |{ COND #( WHEN needs_inner_exact = abap_true THEN
|EXACT #( { rhs } )| ELSE rhs ) } )| ).
  new_statement = break_into_lines( flat_new_statement ).
ENDMETHOD.

METHOD break_into_lines.
  CONSTANTS allowed_line_length TYPE i VALUE 255.
  DATA(remaining_chunk) = strlen( code ).
  WHILE remaining_chunk > 0.
    DATA(already_chopped_chars) = lines( code_lines ) * allowed_line_length.
    DATA(chars_to_chop) = COND #( WHEN remaining_chunk > allowed_line_length THEN
allowed_line_length ELSE remaining_chunk ).
    INSERT code+already_chopped_chars(chars_to_chop) INTO TABLE code_lines.
    remaining_chunk -= chars_to_chop.
  ENDWHILE.
ENDMETHOD.

METHOD build_compute_quickfixes.
  quickfixes = cl_ci_quickfix_creation=>create_quickfix_alternatives( ).
  add_delete_compute_fix(
    EXPORTING procedure = procedure
              statement = statement
    CHANGING quickfixes = quickfixes
  ).
ENDMETHOD.

METHOD add_delete_compute_fix.
  DATA(fix) = CAST if_ci_quickfix_abap_actions( quickfixes->create_quickfix( ) ).
  fix->replace_by(
    p_new_code = ``
    p_context  = cl_ci_quickfix_abap_context=>create_from_comp_procs_tokens(
      p_action_stmt = statement
      p_proc_def = procedure
      p_proc_defs = proc_defs
      p_from_token = 1
    )
  ).
  ASSIGN statement-tokens[ 2 ] TO FIELD-SYMBOL(<second_token>).
  IF <second_token>-refs IS INITIAL AND <second_token>-str = 'EXACT'.
    fix->replace_by(
      p_new_code = ``
      p_context = cl_ci_quickfix_abap_context=>create_from_comp_procs_tokens(
        p_action_stmt = statement
```

```
        p_proc_def = procedure
        p_proc_defs = proc_defs
        p_from_token = 2
      )
    ).
    DATA(rhs_context) = cl_ci_quickfix_abap_context=>create_from_comp_procs_tokens(
      p_proc_defs = proc_defs
      p_proc_def = procedure
      p_action_stmt = statement
      p_from_token = 5
      p_to_token = lines( statement-tokens )
    ).
    fix->insert_before(
      p_new_code = `EXACT #(`
      p_context = rhs_context
    ).
    fix->insert_after(
      p_new_code = `)`
      p_context = rhs_context
    ).
  ENDIF.
  cast if_ci_quickfix_single( fix )->add_docu_from_msgclass(
    p_msg_class = quickfix_docu_message_class
    p_msg_number = 002
  ).
  CAST if_ci_quickfix_single( fix )->enable_automatic_execution( ).
ENDMETHOD.


ENDCLASS.
```

Unit tests:

```
CLASS simple_moves DEFINITION FINAL FOR TESTING
  DURATION SHORT
  INHERITING FROM cl_ci_test_verify
  RISK LEVEL HARMLESS.

PRIVATE SECTION.
  CONSTANTS cut_name TYPE sci_chk VALUE 'CL_CI_TEST_DOCU_EXAMPLE'.

  TYPES ty_expected_quickfix TYPE STANDARD TABLE OF string WITH EMPTY KEY.

  METHODS move_report FOR TESTING RAISING cx_static_check.
  METHODS compute_report FOR TESTING RAISING cx_static_check.
  METHODS assert_finding
    IMPORTING finding_code TYPE sci_errc
              position TYPE cl_ci_test_docu_example=>ty_position
              expected_quickfixed_code TYPE ty_expected_quickfix
    RAISING cx_static_check.
ENDCLASS.


CLASS simple_moves IMPLEMENTATION.

METHOD move_report.
  DATA(test_report) = CONV sobj_name( 'RS_CI_TEST_DOCU_EXAMPLE_1' ).
  run(
    p_variant  = 'DOCU_EXAMPLE'
```

```
    p_obj_type = 'PROG'
    p_obj_name = test_report
    p_create_quickfixes = abap_true
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 23 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `lhs = rhs.` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 26 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `lhs = lines( tab_of_int ).` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 28 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `lhs = lcl=>return_int( ).` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 29 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `lhs = lcl=>return_int_from_two_pars( par_1 = lhs par_2 = lhs ).` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 38 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `lhs_struct = CORRESPONDING #( BASE ( lhs_struct ) rhs_struct ).` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 39 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `lhs = EXACT #( rhs ).` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 48 column = 4 )
    expected_quickfixed_code = VALUE #(
      ( `sub = CAST #( sup ).` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-simple_move
    position     = VALUE #( include = test_report line = 49 column = 4 )
    expected_quickfixed_code = VALUE #(
```

```
      ( `lhs_struct = CORRESPONDING #( BASE ( lhs_struct ) EXACT #( rhs_struct ) ).` )
    )
  ).
ENDMETHOD.

METHOD compute_report.
  DATA(test_report) = CONV sobj_name( 'RS_CI_TEST_DOCU_EXAMPLE_2' ).
  run(
    p_variant  = 'DOCU_EXAMPLE'
    p_obj_type = 'PROG'
    p_obj_name = test_report
    p_create_quickfixes = abap_true
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-compute
    position     = VALUE #( include = test_report line = 8 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `var_1 = var_2 + var_3.` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-compute
    position     = VALUE #( include = test_report line = 10 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `var_1 = EXACT #( var_2 + var_3 ).` )
    )
  ).
  assert_finding(
    finding_code = cl_ci_test_docu_example=>finding_codes-compute
    position     = VALUE #( include = test_report line = 22 column = 2 )
    expected_quickfixed_code = VALUE #(
      ( `sub ?= sup.` )
    )
  ).
ENDMETHOD.


METHOD assert_finding.
  DATA(error_message) = check(
    p_test      = cut_name
    p_code      = finding_code
    p_sobjtype  = 'PROG'
    p_sobjnname = position-include
    p_line      = position-line
    p_col       = position-column
  ).
  IF error_message IS NOT INITIAL.
    cl_abap_unit_assert=>fail(
      msg = error_message
      quit = if_abap_unit_constant=>quit-no
    ).
  ELSE.
    DATA(quickfix_tester) = NEW cl_ci_quickfix_testing( ).
    quickfix_tester->set( p_detail = result_list[
       sobjtype = 'PROG'
       sobjname = position-include
       line = CONV #( position-line )
```

```
      col = CONV #( position-column )
    ]-detail ).
    DATA(errors) = VALUE cl_ci_quickfix_testing=>t_err_msgs( ).
    DATA(quickfixed_sources) = VALUE cl_ci_quickfix_testing=>t_result_sources( ).
    DATA(quickfix_okay) = quickfix_tester->check_quickfixes(
      EXPORTING
        p_program = position-include
        p_expected_quickfixes = VALUE #( (
          qf_code      = 'QFIX1'
          replacements = VALUE #( (
            include   = position-include
            from_line = position-line
            to_line   = position-line + lines( expected_quickfixed_code ) - 1
            by        = CONV #( expected_quickfixed_code )
          ) )
        ) )
      IMPORTING
        p_err_msgs = errors
        p_result_sources = quickfixed_sources
    ).
    IF quickfix_okay = abap_false.
      DATA(first_error) = errors[ 1 ].
      cl_abap_unit_assert=>assert_true(
        act = quickfix_okay
        msg = |Quickfix error: { first_error-msg }. Erroneous line: {
quickfixed_sources[ include = first_error-include ]-source[ first_error-line ] }|
        quit = if_abap_unit_constant=>quit-no
      ).
    ENDIF.
  ENDIF.
ENDMETHOD.


ENDCLASS.
```

Message class CI_DOCU_EXAMPLE

001 Replace MOVE by assignment operator (Long text: This quickfix replaces a MOVE statement by the equivalent modern version using the assignment operator '='.)

002 Delete COMPUTE

First test report

```
REPORT rs_ci_test_docu_example_1.

CLASS lcl DEFINITION.
PUBLIC SECTION.
  CLASS-METHODS return_int
    RETURNING VALUE(an_int) TYPE i.
  CLASS-METHODS return_int_from_two_pars
    IMPORTING par_1 TYPE i
              par_2 TYPE i
    RETURNING VALUE(an_int) TYPE i.
ENDCLASS.

CLASS lcl IMPLEMENTATION.
```

```
  METHOD return_int.
  ENDMETHOD.
  METHOD return_int_from_two_pars.
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  DATA lhs TYPE i.
  DATA rhs TYPE i.
  MOVE rhs TO lhs.

  DATA tab_of_int TYPE STANDARD TABLE OF i.
  MOVE lines( tab_of_int ) TO lhs.

  MOVE lcl=>return_int( ) TO lhs.
  MOVE lcl=>return_int_from_two_pars( par_1 = lhs par_2 = lhs ) TO lhs.

  TYPES:
    BEGIN OF ty_struct,
       comp_1 TYPE i,
       comp_2 TYPE i,
    END OF ty_struct.
  DATA rhs_struct TYPE ty_struct.
  DATA lhs_struct TYPE ty_struct.
  MOVE-CORRESPONDING rhs_struct TO lhs_struct.
  MOVE EXACT rhs TO lhs.

  CLASS sub_lcl DEFINITION
      INHERITING FROM lcl.
  ENDCLASS.

  START-OF-SELECTION.
    DATA sub TYPE REF TO sub_lcl.
    DATA sup TYPE REF TO lcl.
    MOVE sup ?TO sub.

    MOVE-CORRESPONDING EXACT rhs_struct TO lhs_struct.
```

Second test report

```
REPORT rs_ci_test_docu_example_2.


START-OF-SELECTION.
  DATA var_1 TYPE i.
  DATA var_2 TYPE i.
  DATA var_3 TYPE i.
  COMPUTE var_1 = var_2 + var_3.
  var_1 = var_2 + var_3.
  COMPUTE EXACT var_1 = var_2 + var_3.

CLASS cl_sup DEFINITION.
ENDCLASS.

CLASS cl_sub DEFINITION
    INHERITING FROM cl_sup.
```

```
ENDCLASS.

END-OF-SELECTION.
  DATA sup TYPE REF TO cl_sup.
  DATA sub TYPE REF TO cl_sub.

  COMPUTE sub ?= sup.
```

www.sap.com/contactsap

© 2018 SAP SE or an SAP affiliate company. All rights reserved.
No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

The information contained herein may be changed without prior notice. Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

In particular, SAP SE or its affiliated companies have no obligation to pursue any course of business outlined in this document or any related presentation, or to develop or release any functionality mentioned therein. This document, or any related presentation, and SAP SE's or its affiliated companies' strategy and possible future developments, products, and/or platform directions and functionality are all subject to change and may be changed by SAP SE or its affiliated companies at any time for any reason without notice. The information in this document is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. All forward-looking statements are subject to various risks and uncertainties that could cause actual results to differ materially from expectations. Readers are cautioned not to place undue reliance on these forward-looking statements, and they should not be relied upon in making purchasing decisions.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies. See http://www.sap.com/corporate-en/legal/copyright/index.epx for additional trademark information and notices.

THE BEST RUN SAP